Faculté des sciences
Aix Marseille Université

Année 2024-25

Domaine Sciences et Technologies
LICENCE MATHÉMATIQUES-INFORMATIQUE
Compilation : TP 1
Codes UE : SIN6U64L
TP 1 : ANTLR

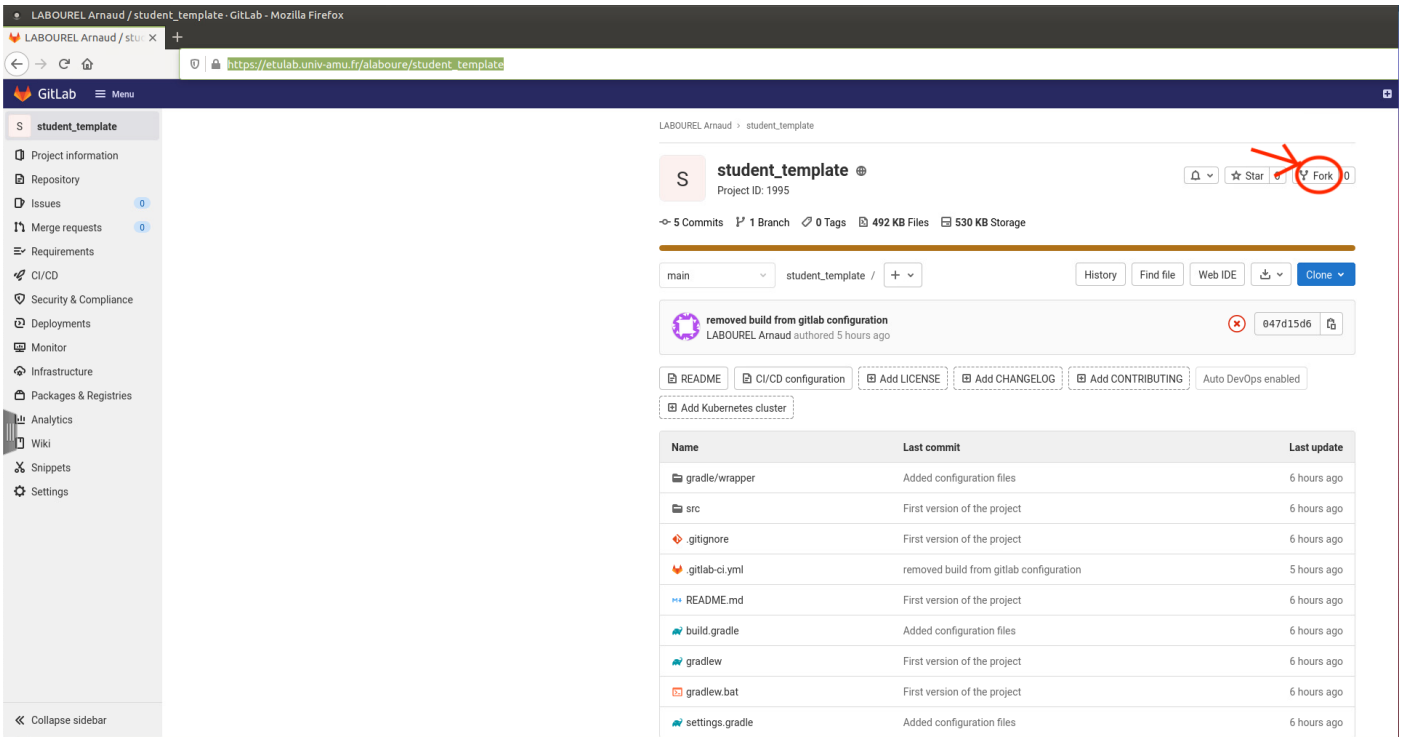
1 Introduction

Le but de ce TP est d'apprendre à utiliser l'analyseur ANTLR pour faire l'analyse lexicale et syntaxique d'entrées à partir d'un lexique et d'une grammaire définis formellement. Vous pouvez faire ce TP seul ou en binôme.

1.1 Fork d'un projet

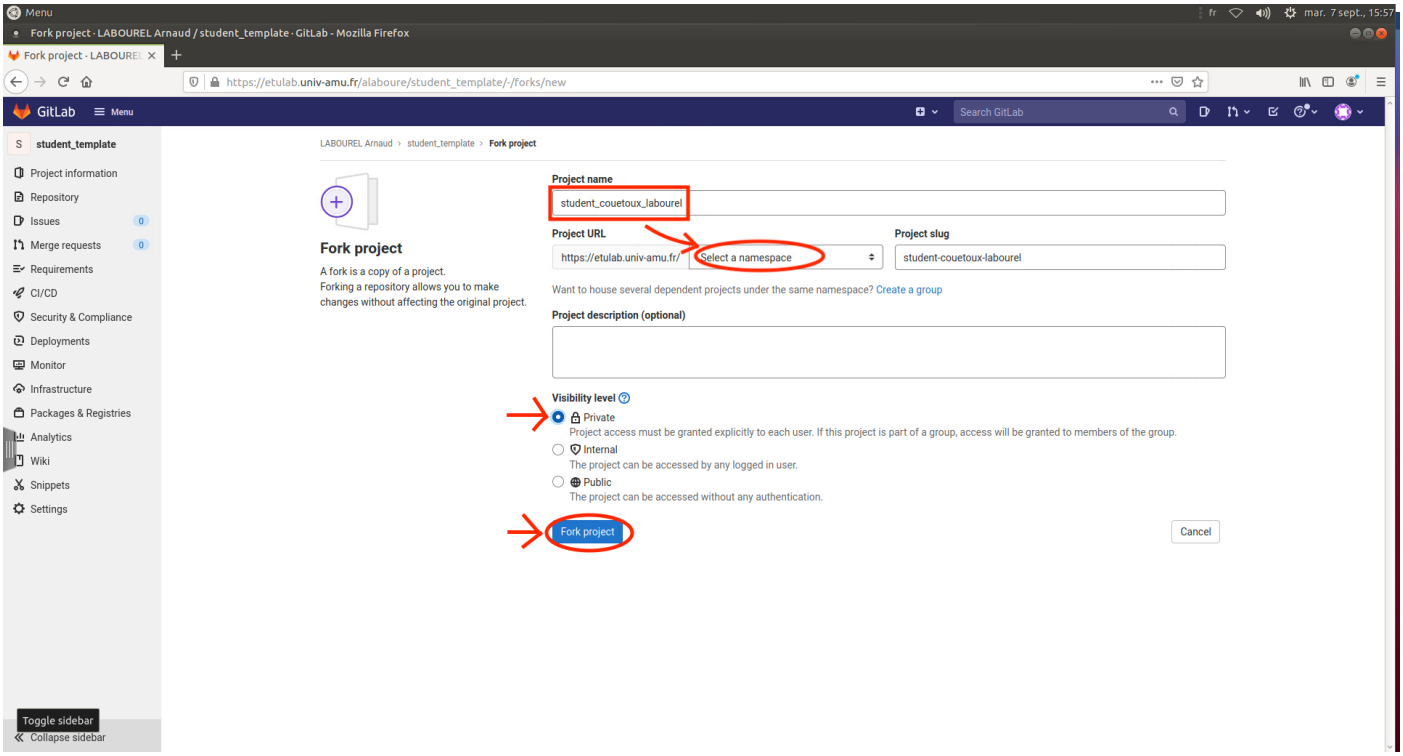
Vous allez maintenant créer votre projet en utilisant un projet déjà existant. Pour cela, il faut :

1. Aller sur le projet `ANTLR introduction` qui servira de base pour ce TP en accédant à l'adresse suivante : https://etulab.univ-amu.fr/alaboure/antlr-introduction
2. Cliquer sur le bouton *fork*.

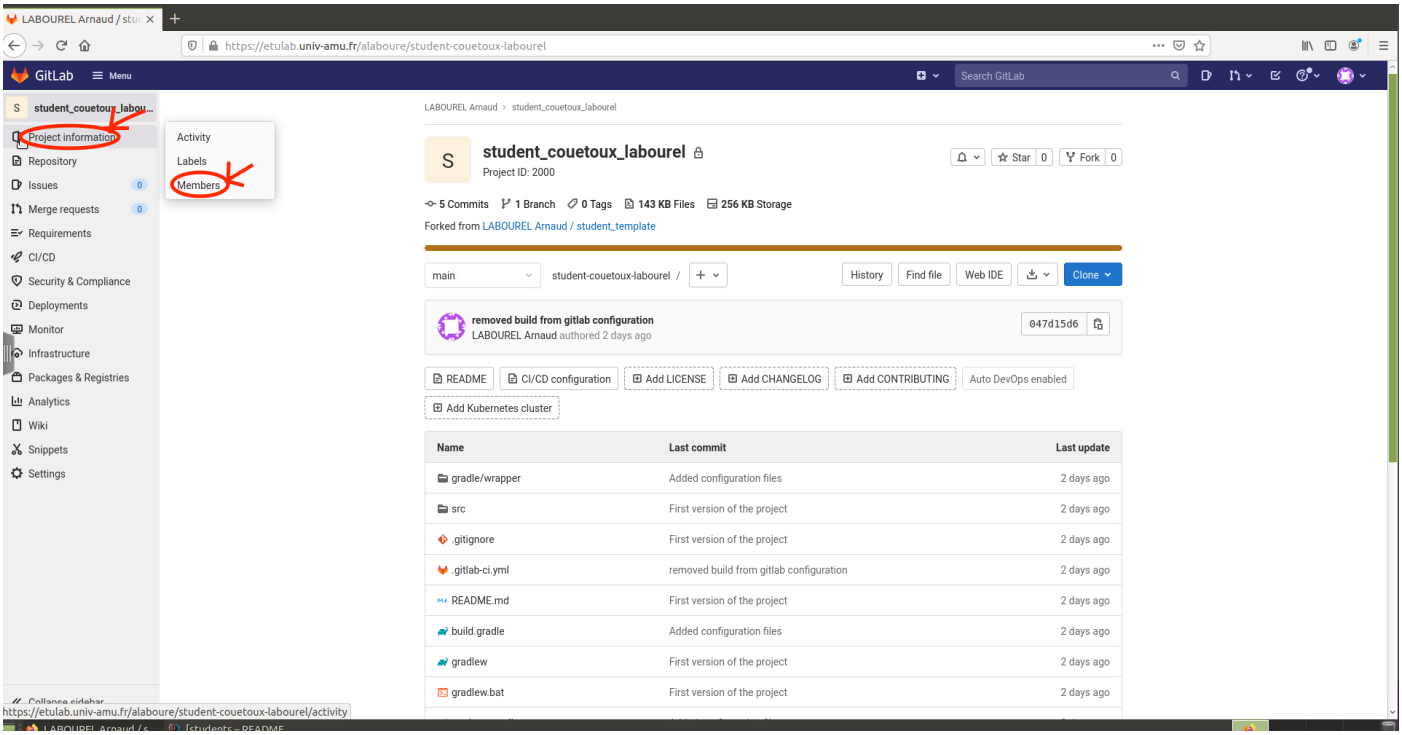


1. Changer le nom du projet pour le changer `antlr-introduction-xxxxx-yyyyy` avec `xxxxx` le nom de famille du premier (par ordre alphabétique) étudiant du binôme et `yyyyy` le nom du deuxième étudiant du binôme si vous faites le sujet en binôme. Si vous faites le projet seul, mettez `antlr-introduction-xxxxx` avec `xxxxx` votre nom de famille.

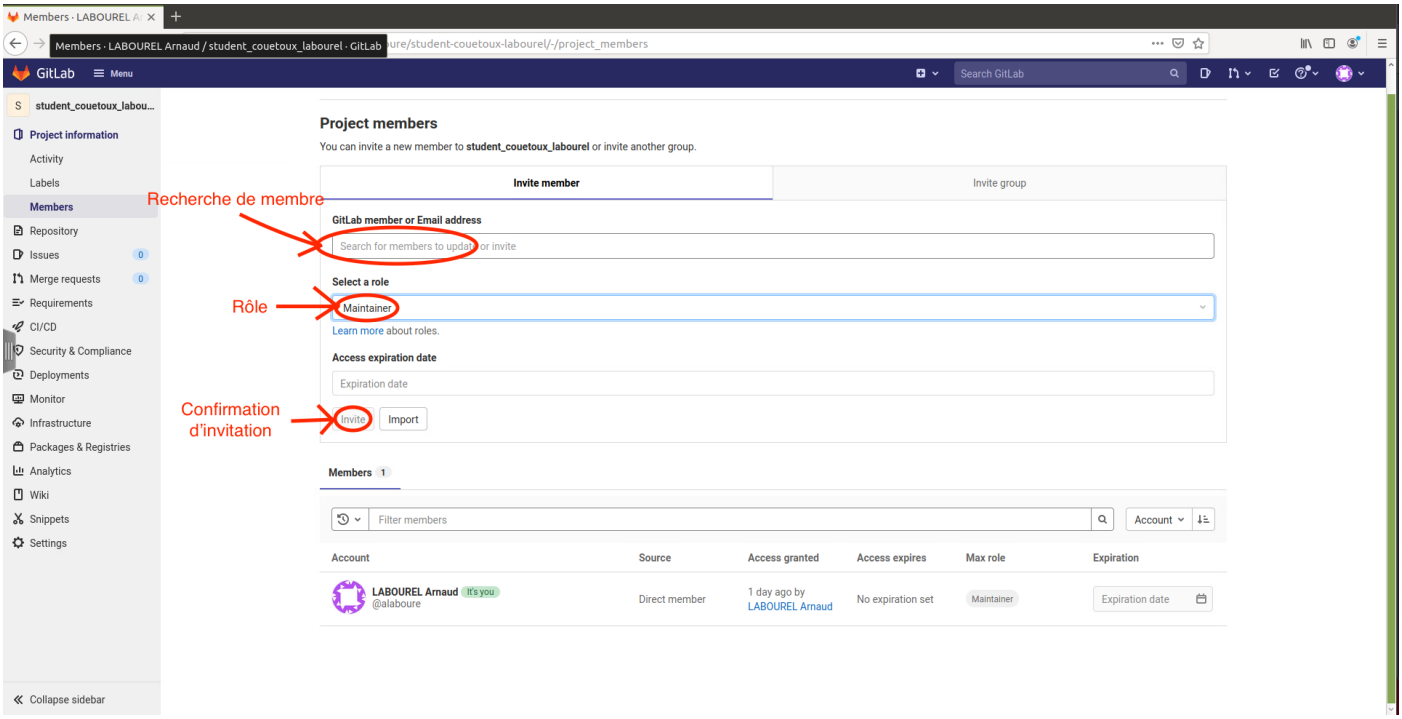
2. Sélectionner comme espace de nom (*spacename*) votre propre compte. Mettez la visibilité du projet en *private* afin que vos camarades en dehors du projet n'y aient pas accès et validez le fork en cliquant sur le bouton `fork project`.



1. Une fois le projet créé, vous pouvez rajouter des membres en cliquant sur `project information` dans le menu de gauche puis `members`.



5. Ensuite vous pouvez rechercher une personne dans la barre dédiée. Une fois celle-ci trouvée vous pouvez lui donner un rôle (au moins `reporter` pour donner l'accès en lecture du code, `maintainer` pour l'accès en lecture et écriture sur la branche principale *master* ou *main* et `owner` pour donner approximativement les mêmes droits que le créateur du projet), puis confirmer son invitation en cliquant sur le bouton `invite`.



2 Explication ANTLR

ANTLR (*ANother Tool for Language Recognition*) est un générateur d'analyseurs pour lire, traiter, exécuter ou traduire du texte structuré ou des fichiers binaires. Il est largement utilisé pour créer des langages, des outils et des frameworks. À partir d'une description de langage formel sous forme de règles de grammaire, ANTLR génère un analyseur pour ce langage défini par cette grammaire. L'analyseur permet à partir d'une entrée de générer l'arbre de dérivation de la grammaire si l'entrée fait partie du langage, mais aussi de calculer des attributs dans le cas d'une grammaire attribuée. ANTLR fournit aussi une interface pour afficher les arbres de dérivation et aussi une interface à implémenter pour visiter les nœuds de ces arbres afin d'exécuter du code spécifique à l'application.

ANTLR permet donc de réaliser deux étapes importantes de la compilation :

— l'analyse lexicale qui génère le flux de *tokens* à partir du flux de caractères ;
— l'analyse syntaxique qui génère l'arbre de dérivation à partir du flux de *tokens*.

2.1 Analyse lexicale

L'analyse lexicale, le *lexing* ou la tokenisation est le processus de conversion d'une séquence de caractères en une séquence de *tokens* (chaînes avec une signification attribuée et donc identifiée). Un programme qui effectue une analyse lexicale est appelé un *lexer*.

ANTLR permet de définir des règles pour les *tokens* sous la forme suivante :

```
NOM_DU_TOKEN : expressionRégulière ;
```

Le nom du token est le nom qu'on donne au *token* qui sera ensuite utilisé ensuite pour l'analyse syntaxique. Le nom du *token* doit commencer par une majuscule (règle à respecter pour les différencier symboles non-terminaux qui seront définis dans la grammaire). L'expression régulière sert à définir le motif qui sera utilisé pour détecter la présence du *token* dans l'entrée.

— `( exp )` : parenthésage de l'expression régulière `exp` ;
— `exp1 | exp2` : union entre deux expressions régulières `exp1` et `exp2` ;
— `'s'` : caractère ou chaîne de caractères `s` (`'\n'` pour le saut de ligne) ;
— `.` : n'importe quel caractère ;
— `a .. b` ou `[a-b]`: caractère entre `a` et `b` (`'0'..'9'` pour un chiffre) ;
— `+` : 1 ou plus (`('0' .. '9')+` pour une séquence d'au moins un chiffre) ;
— `*` : 0 ou plus (`('0' .. '9')*` pour une séquence de chiffres potentiellement vide) ;
— `?` : optionnel (`[0-9]+ ('.' [0-9]*)?` pour un littéral correspondant à un `float`) ;
— `[abc]` : choix entre caractères `a`, `b` ou `c`
— `~[...]` : complémentaire (par exemple `~[abc]` pour un caractère autre que `a`, `b` ou `c` ;
— `// ...` et `/* ... */` : commentaires.

Exemples de règles de *tokens* :

```
HELLO : 'hello' ; // un mot fixe
ID : [a-z]+ ; // un mot composé de lettres minuscules
INT : [0-9]+ ; // un mot composé de chiffres
WS : [ \t\r\n]+ -> skip ; // une règle pour ignorer les espaces, tabulations et sauts de lignes.
```

Dans le cas où plusieurs règles pourraient s'appliquer en même temps le *lexer* applique d'abord la règle permettant de créer le *token* le plus long et en cas d'égalité de longueur la règle définie en premier dans le fichier (règle commune dans quasiment tous les *lexers*).

Le fichier de règles du *lexer* doit commencer par une ligne donnant le nom du *lexer* qui est au format suivant :

```
lexer grammar NomDuLexer;
```

Le nom doit correspondre au nom du fichier auquel on rajoute le suffixe `.g4` ce qui nous donne `NomDuLexer.g4`. Dans la configuration du dépôt, il est important que le fichier soit dans le répertoire `src/main/antlr` afin que la génération des classes à partir du fichier `.g4` se fasse correctement.

Si on souhaite que les classes générées par ANTLR pour le *lexer* soient dans un package, on peut rajouter les lignes suivantes :

```
@header {
package nom.du.package;
}
```

Vous trouverez davantage de détails sur les règles du *lexer* d'ANTLR au lien suivant : https://github.com/antlr/antlr4/blob/master/doc/lexer-rules.md.

Le fichier `SampleLexer.g4` dans le répertoire `src/main/antlr` du dépôt contient un fichier de description pour un *lexer*. Ce fichier de description permet à ANTLR de construire un fichier `SampleLexer.java` qui définit une classe étendant la classe `Lexer`. On peut ensuite utiliser cette classe pour réaliser l'analyse lexicale d'une entrée. La classe `Main` dans le dossier `src/main/sample/lexer` du dépôt vous donne un exemple d'utilisation du *lexer* généré à partir du fichier `SampleLexer.g4`. La première ligne du code de la méthode `main` de cette classe permet de créer un stream de caractères à partir d'une chaîne de caractères qui sera l'entrée du *lexer* :

```
CharStream stream =
  CharStreams.fromString("{if (x ==1.2) then {x=512;} else {x=645;} if (y ==2) then {y=512;}}");
```

La deuxième ligne du code permet de créer un objet correspondant au *lexer* :

```
sample.lexer.SampleLexer lexer = new sample.lexer.SampleLexer(stream);
```

Finalement, la boucle `for` suivante permet de parcourir les *tokens* pour les afficher :

```
for (Token token = lexer.nextToken();
     token.getType() != Token.EOF;
     token = lexer.nextToken()) {
```

```
  printToken(token, lexer);
}
```

2.2 Questions sur l'analyse lexicale

1. Lancez le *lexer* via la méthode `main` de la classe `sample.lexer.Main`. Quels sont les *tokens* générés par le *lexer* ?
2. Ouvrez le fichier `SampleLexer.java` dans le répertoire `build/generated-src/antlr/main`. D'après vous à quoi sert ce fichier ? Quel est le sens derrière l'acronyme `DFA` qui est utilisé comme type d'élément d'un attribut tableau ?
3. Ouvrez le fichier `SampleLexer.g4` dans IntelliJ. Est-ce qu'il y a une coloration syntaxique des éléments de la grammaire ?
4. Installer le plugin *ANTLR* pour IntelliJ IDEA. Pour cela, vous pouvez aller dans le menu aux préférences (`file - > settings` sur linux), choisir `plugin` dans la colonne à gauche de la fenêtre qui s'est ouverte, rechercher `antlr` et finalement installer le plugin nommé `ANTLR v4`. IntelliJ IDEA a besoin d'être redémarré et va normalement vous proposer le redémarrage. Ouvrez à nouveau le fichier `SampleLexer.g4` dans IntelliJ IDEA après le redémarrage de l'application. Est-ce qu'il y a maintenant une coloration syntaxique des éléments de la grammaire ?
5. Modifiez le fichier `SampleLexer.g4` de sorte à que la ligne du token `ID` soit avant celle du token `ELSE` puis relancez le programme. Quels sont les différences dans les *tokens* générés par le *lexer* ? Comment expliquez-vous cette différence ? Remettez pour la suite du TP les deux lignes dans l'ordre de départ.
6. Modifiez la `String` en entrée du programme en remplaçant les `x` par `x1` et donc en remplaçant la première ligne de la méthode `main` par la ligne suivante :

```
CharStream stream = CharStreams.fromString(
              "{if (x1 ==1.2) then {x1=512;} else {x1=645;} if (y ==2) then {y=512;}}");
```

Quels sont les différences dans les *tokens* générés par le *lexer* ? Est-ce que cela correspond selon vous au comportement souhaitable du *lexer* ? Modifiez les règles contenues dans le fichier de sorte à qu'un *token* de type `ID` corresponde à une séquence d'un ou plusieurs caractères (lettres et chiffres) dont le premier est obligatoirement une lettre.

7. Rajouter des règles pour détecter les opérateurs suivants et mot-clés suivants : `<`, `>`, `<=`, `>=`, `&&`, `||`, `!`, `while`, `for`, `do`.

2.3 Analyse syntaxique

ANTLR permet de réaliser l'analyse syntaxique qui génère l'arbre de dérivation à partir du flux de *tokens* obtenu du *lexer*. On appelle *parser* le programme généré par ANTLR qui permet de construire l'arbre de dérivation.

Le format de fichier est assez similaire au format pour le *lexer*. Les différences sont les suivantes :

— Le fichier commence par une ligne indiquant le nom du *parser* et qui commence par `parser` au lieu de `lexer`

```
parser grammar NomDuParser;
```

— Il est nécessaire de donner le nom du *lexer* qui produira les *tokens* utilisés par le *parser* avec la ligne suivante :

```
options {tokenVocab=NomDuLexer;}
```

Il faudra dans ce cas que le fichier `NomDuLexer.g4` définisse les *tokens* utilisés par la grammaire.

— Le format des règles hors contextes a deux différences avec les règles de *lexer* :

— le nom du symbole non-terminal doit commencer par une minuscule :

```
symbol : 'a' | 'b';
```

— la partie droite de la règle peut faire référence à des symboles non-terminaux ou des *tokens* (en plus d'expression régulière classique à base de caractères terminaux)

```
instructionBlock : OPENING_CURLY_BRACKET instruction CLOSING_CURLY_BRACKET;
instruction : ID ASSIGN literal SEMICOLON;
```

ANTLR admet les séquences de symboles non-terminaux dans la partie droite de la règle :

```
prog: statement+ ; // one or more statements
block: statement* ; // zero or more statements
```

Le fichier `SampleParser.g4` dans le répertoire `src/main/antlr` du dépôt contient un fichier de description pour un *parser*. Ce fichier de description permet à ANTLR de construire les fichiers suivants :

— Un fichier `SampleParser.java` qui définit une classe étendant la classe `Parser`. On peut ensuite utiliser cette classe pour réaliser l'analyse syntaxique d'une entrée et donc construire l'arbre de dérivation.

— Une interface `SampleParserVisitor<T>` étendant l'interface `ParseTreeVisitor<T>` qui permet de visiter les nœuds de l'arbre de dérivation.

— Une classe `SampleParserBaseVisitor<T>` étendant la classe `AbstractParseTreeVisitor<T>` qui visite les tous nœuds de l'arbre en appelant `visit` sur les fils du nœud sans réaliser d'autres opérations.

La classe `Main` dans le dossier `src/main/sample/parser` du dépôt vous donne un exemple d'utilisation du *parser* généré à partir du fichier `SampleParser.g4`. Il est important que ce fichier se trouve dans le répertoire `src/main/antlr` du dépôt. La première ligne du code de la méthode `main` de cette classe permet de créer un stream de caractères à partir d'une chaîne de caractères qui sera l'entrée du *lexer* :

```
CharStream stream =
  CharStreams.fromString("{if (x ==1.2) then {x=512;} else {x=645;} if (y ==2) then {y=512;}}");
```

La deuxième ligne du code permet de créer un objet correspondant au *lexer* :

```
SampleLexer lexer = new SampleLexer(stream);
```

La troisième ligne permet de construire l'objet correspondant au flux de *tokens* générés par le *lexer* :

```
TokenStream tokenStream = new CommonTokenStream(lexer);
```

La quatrième ligne permet de construire l'objet correspondant au *parser* à partir du flux de tokens généré par le *lexer* :

```
SampleParser parser = new SampleParser(tokenStream);
```

La cinquième ligne permet de construire l'arbre de dérivation :

```
ParseTree tree = parser.instructionBlock();
```

La méthode appelée sur l'objet *parser* doit avoir le même nom que le symbole non-terminal axiome (constituant le proto-mot de départ utilisé pour l'analyse).

Une fois l'arbre construit, on peut :

— l'afficher sous forme de chaîne de caractères avec le format `(parent enfant1 enfant2 ... )` :

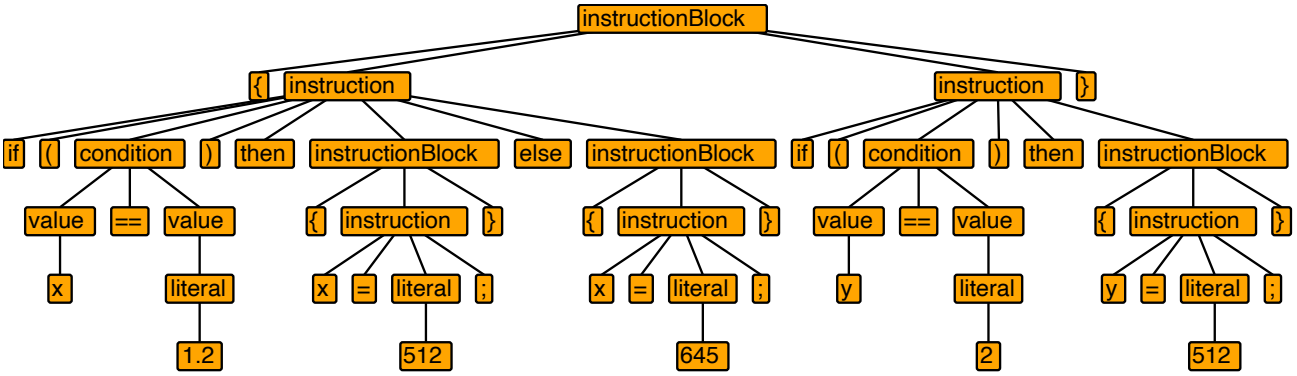
```
System.out.println(tree.toStringTree(parser));
```

— l'afficher grâce à l'interface graphique d'ANTLR :

```
TreeViewer viewer = new TreeViewer(Arrays.asList(parser.getRuleNames()), tree);
viewer.open();
```

2.4 Questions sur l'analyse syntaxique

1. Lancez le *parser* via la méthode `main` de la classe `sample.parser.Main`. Vous devriez obtenir l'arbre de dérivation suivant :



2. Modifier la grammaire de `SampleParser.g4` pour rajouter des règles pour gérer :

— les conditions avec les comparateurs `<`, `>`, `<=` et `>=` ;
— les conditions avec opérateurs logiques `||`, `&&` et `!`.
— les boucles `for` ;
— les boucles `while` et `do-while`.

2.5 Grammaires attribuées

Il est possible de définir dans ANTLR des attributs aux nœuds de l'arbre de dérivation. Comme on a vu en cours, il y a deux types d'attributs :

— les attributs hérités dont la valeur à chaque nœud dépend de la valeur dans le nœud parent dans l'arbre de dérivation ;
— les attributs synthétisés dont la valeur à chaque nœud dépend des valeurs des nœuds enfants dans l'arbre de dérivation.

2.5.1 Attributs hérités

En ANTLR, on peut définir les attributs hérités d'un symbole non-terminal au moment de la définition de la règle avec le format suivant :

```
nomSymbole1 [typeAttributHérité nomAttributHérité] :
  nomSymbole2[valeurAttribut2] nomSymbole3[valeurAttribut3];
nomSymbole2 [typeAttributHérité nomAttributHérité] : ...
nomSymbole3 [typeAttributHérité nomAttributHérité] : ...
```

Les attributs hérités ont un nom et un type (dans notre cas un type `Java`). `valeurAttribut2` et `valeurAttribut3` correspondent à des expressions qui donnent les valeurs des attributs des symboles `nomSymbole2` et `nomSymbole3`. Un attribut du symbole réécrit par la règle est accessible via `$nomAttributHérité`. Pour les autres symboles, les attributs sont accessibles via `$nomSymbole.nomAttributHérité`. À noter que l'on doit donner une valeur à l'attribut hérité de l'axiome lorsqu'on crée l'arbre dans le code :

```
ParseTree tree = parser.nomAxiome(valeurAttributAxiome);
```

Par exemple, si on crée une grammaire pour un mot (`word`) composé de lettres et que l'on souhaite avoir un attribut hérité correspondant à la taille de la police du texte (`size`) qui aura la même valeur pour tous les nœuds de l'arbre de dérivation, on peut définir la grammaire suivante :

```
word [double size] : character[$size]
                   | character[$size] word[$size];
character [double size] : CHARACTER

CHARACTER : [a-zA-Z];
```

Pour faire l'analyse syntaxique avec ces règles en initialisant la `size` de la racine à `10`, il suffit d'écrire la ligne suivante :

```
ParseTree tree = parser.word(10);
```

2.5.2 Attributs synthétisés

Pour les attributs synthétisés, on utilise la syntaxe suivante :

```
nomDeSymbole1 returns [typeAttributSynthétisé nomAttributSynthétisé]
                : nomDeSymbole2 nomDeSymbole3;
                  {$nomAttributSynthétisé1 = valeurAttribut1; instruction; };
nomDeSymbole2 returns [typeAttributSynthétisé nomAttributSynthétisé] :
nomDeSymbole3 returns [typeAttributSynthétisé nomAttributSynthétisé] :
```

Les attributs synthétisés ont un nom et un type (dans notre cas un type `Java`). `valeurAttribut1` correspond à des expressions qui donne la valeur de l'attribut `nomAttributSynthétisé1` et qui peut utiliser les attributs des deux autres symboles. `instruction` correspond à du code Java quelconque. Dans le cas ou une règle contiendrait plusieurs fois le même symbole, on peut leur donner un nom en utilisant `nomDanslaRègle=NomDuSymbole`. Par exemple, on peut utiliser la règle suivante pour calculer la valeur d'une addition pour une expression :

```
expr returns [ double value ] : e1=expr `+` e2=expr { $value = $e1.value + $e2.value; };
```

2.6 Exemple d'application de grammaire attribuée

Le fichier `Typography.g4` dans le répertoire `src/main/antlr` définit une variante de la grammaire vue en cours pour la typographie. Les règles sont les suivantes :

Productions	Règles sémantiques
$S \to B$	$B.size = 10$
$B \to CB$	$B_1.size = C.size = B.size$ $B.height = \max(B_1.height, C.heigth)$ $B.depth = \max(B_1.depth, C.depth)$
$B \to C\_\{B\}B$	$B_1.size = B.size$ $B_2.size = \frac{7}{10}B.size$ $B.height = \max(B_1.height, B_2.height, C.height - \frac{1}{4}B.size)$ (la ligne est abaissée) $B.depth = \max(B_1.depth, B_2.depth, C.depth + \frac{1}{4}B.size)$
$B \to C$	$C.size = B.size, B.height = C.height, B.depth = C.depth$
$B \to \varepsilon$	$B.height = 0$ $B.depth = 0$
$C \to \alpha$	$C.height = \text{heightOf}(\alpha, C.size)$ (α étant un caractère) $C.depth = \text{depthOf}(\alpha, C.size)$

Généralement lorsqu'on écrit une grammaire à la main les symboles non-terminaux sont des lettres majuscules, mais il est plus propre de définir de véritables noms pour les symboles non-terminaux des grammaires définies via le format d'ANTLR. On a donc utilisé les noms suivants :

— S devient `program`

— B devient `box`
— C devient `character`

La classe `TypographyMain` dans le répertoire `src/main/java/attributes` permet de lancer l'analyse. On peut remarquer quelques différences avec l'analyse précédente :

— Les règles du *lexer* et *parser* sont définies dans le même fichier `Typography.g4` qui génère un *lexer* `TypographyLexer` et un *parser* `TypographyParser`. Le fichier commence par la ligne `grammar Typography` qui permet de définir en un seul fichier *lexer* et *parser*. De manière générale, un fichier `NomDeGrammaire` commençant par `grammar NomDeGrammaire` génère un *lexer* `NomDeGrammaireLexer` et un *parser* `NomDeGrammaireParser`.
— L'analyse se lance à partir du contenu d'un fichier au lieu d'une simple chaîne de caractères. La ligne permettant de générer le *stream* (flux) de caractères servant d'entrée au *lexer* est :

```
CharStream stream = CharStreams
      .fromStream(Objects.requireNonNull(ClassLoader.getSystemResourceAsStream("text")));
```

Cette ligne ouvre le fichier `text` dans le répertoire `src/main/resources` du projet. Il est important que le fichier se trouve dans le dossier `resources` pour que cela fonctionne. Cela demande aussi de gérer (capturer avec le mot-clé `catch`) les exception de `Exception`
— Grâce à `@header`, on a rajouté des `import` permettant de rajouter des dépendances qui pourront être utilisé dans le code associé aux règles de la grammaire.

```
@header{
import sample.attributes.Characters;
import sample.attributes.Printer;
}
```

Ici, cela nous permet de faire des références aux classes `Characters` et `Printer` présentes dans le répertoire `src/main/java/sample/attributes`.
— Grâce à `@header`, on peut initialiser des variables globales d'analyse qui seront accessibles durant la création de l'arbre et l'exécution du code associé aux règles. Ici, cela nous permet d'initialiser un `Printer` utilisé pour afficher les valeurs des attributs des nœuds de l'arbre avec les lignes suivantes :

```
@members{
Printer printer = new Printer();
}
```

2.7 Questions sur les grammaires attribuées

1. Tester l'analyse produite sur l'entrée de base et vérifier que les règles de calcul des attributs sont bien appliquées. Modifier l'entrée pour qu'elle ne soit plus valide (ne fasse plus partie du langage défini par la grammaire) et relancer l'analyse.
2. Rajouter dans le fichier `Typography.g4` un attribut synthétisé `width` correspondant à la largeur de la boîte. On utilisera les règles suivantes pour le calcul de la largeur d'une boîte :
   — La largeur d'un caractère est égal à 60% de sa taille (par exemple un caractère de taille 10 aura une largeur de 6). La largeur est la même pour tous les caractères (police monospace).
   — La largeur d'une boîte est la somme des largeurs des éléments qui la compose lorsqu'on applique une

règle.

3. Modifier la classe `Printer` et le code des actions sémantiques dans le fichier `Typography.g4` afin d'afficher la largeur de chaque nœud de l'arbre de dérivation.

2.8 Visiteurs d'arbre de dérivation

Bien qu'il soit commode de définir des règles de calcul d'attributs directement dans la grammaire, il est plus propre dans le cas de grammaire complexe de générer l'arbre de dérivation pour ensuite le parcourir. Une manière simple de faire cela est d'utiliser le patron de conception visiteur. Comme cela a déjà été expliqué, ANTLR génère une interface à implémenter pour les visiteurs ainsi qu'un visiteur de base tous les deux paramétrés par le type de retour de la méthode de visite. L'exemple qui vous est donné est un évaluateur d'expression (calcul avec des `+`, `*` et des entiers) dont la grammaire est définie dans le fichier `Expression.g4`. ANTLR génère donc les deux fichiers suivants :

— Une interface `ExpressionVisitor<T>` étendant l'interface `ParseTreeVisitor<T>` qui permet de visiter les nœuds de l'arbre de dérivation.
— Une classe `ExpressionBaseVisitor<T>` étendant la classe `AbstractParseTreeVisitor<T>` qui visite les tous nœuds de l'arbre en appelant `visit` sur les fils du nœud sans réaliser d'autres opérations.

Généralement, il est plus pour faire un visiteur de différencier chaque alternative (façon de réécrire le symbole non-terminal) d'une règle de grammaire. En ANTLR, on peut le faire avec le symbole `#` suivi du nom que l'on souhaite donner à l'alternative. La syntaxe est la suivante :

```
nomDeSymboleTerminal : expressionPourLAlternative1 # nomAlternative1
                     | expressionPourLAlternative2 # nomAlternative2 ;
```

Pour l'évaluation d'une expression on peut donc définir les alternatives d'une règle de la manière suivante :

```
expression: expression PLUS expression # Addition
          |   expression TIMES expression # Multiplication
          |   INT # Integer
          |   OPENING_PARENTHESIS expression CLOSING_PARENTHESIS # InParenthesesExpression ;
```

L'interface pour le visiteur contient dans ce cas une méthode par alternative de règle. Ce qui nous donne donc dans ce cas l'interface suivante :

```
public interface ExpressionVisitor<T> extends ParseTreeVisitor<T> {
    T visitInteger(ExpressionParser.IntegerContext ctx);
    T visitInParenthesesExpression(ExpressionParser.InParenthesesExpressionContext ctx);
    T visitAddition(ExpressionParser.AdditionContext ctx);
    T visitMultiplication(ExpressionParser.MultiplicationContext ctx);
}
```

Le paramètre `ctx` de chaque méthode `visit` correspond au nœud en train d'être visité. On peut accéder à ses fils en appelant sur `ctx` une méthode ayant le même nom que les symboles de la règle qui correspondent soit à des symboles non-terminaux soit à des lexèmes (par exemple dans notre cas `expression()` pour accéder

au fils correspondant à `expression` ou `INT` pour accéder à un fils correspondant à un lexème `INT`). Si la règle contient plusieurs fois le même symbole alors la méthode renvoie une liste correspondant aux fils associé à ce type de symbole. Dans ce cas, il aussi est possible de donner un argument entier correspondant à l'indice du symbole (par exemple, on peut utiliser `expression(1)` pour accéder au nom de l'arbre correspondant au deuxième symbole `expression` de la règle).

Lorsqu'on a un nœud `node` de l'arbre, on peut récupérer son texte avec `node.getText()` ou demander la visite sur le nœud avec `node.accept(this)`.

La classe `EvaluationVisitor` dans le dossier `src/main/java/sample/visitor` définit un visiteur pour la grammaire `Expression`. On commence par créer l'arbre de dérivation avec la ligne suivante :

```
ParseTree tree = parser.expression();
```

On crée ensuite un visiteur :

```
EvaluationVisitor visitor = new EvaluationVisitor();
```

On demande au visiteur de visiter l'arbre et on stocke le résultat calculé dans une variable :

```
Integer value = visitor.visit(tree);
```

2.9 Questions sur les visiteurs

1. Qu'est-ce qui est calculé par le visiteur `EvaluationVisitor` ?
2. Est-ce que ce qui est calculé par le visiteur `EvaluationVisitor` correspond à l'ordre de priorité des opérateurs `*` et `+` ?
3. Corrigez la grammaire (le fichier `Expression.g4`) pour que le calcul corresponde à l'ordre de priorité classique des opérateurs `*` et `+` (`*` étant plus prioritaire que `+`).
4. Créer un visiteur permettant de générer la notation postfixée parenthésée d'une expression. Votre visiteur devra donc produire la chaîne de caractères `(((10, 20, *), 30, +), 1, +)` à partir de l'entrée `10*20+30+1` (n'oubliez pas que vous devez avoir corrigé le problème de priorité des opérateurs avant de répondre à cette question).